TRACE-BASED HARDWARE CONTROL FLOW SIGNATURE CHECKING

BY

ERIC JAMES CLARK

THESIS

Submitted in partial fulfillment of the requirements
for the degree of Master of Science in Electrical and Computer Engineering
in the Graduate College of the
University of Illinois at Urbana-Champaign, 2018

Urbana, Illinois

Adviser:

Associate Professor Michael Bailey

ABSTRACT

Since the widespread adoption of the internet, computer security has become one of the primary concerns of computer systems engineers, who are often faced with a choice between increased performance and stronger security policies. Many advancements in computer security impose a performance penalty. One such advancement, control-flow integrity (CFI), offers immunity to code-reuse attacks but slows down software that enforces CFI. In this work, we present a hardware monitor design that utilizes hardware support for instruction tracing to detect code-reuse attacks. The monitor is small in hardware size, uses low power, and does not decrease software performance.

In memory of Maximillian Joseph Clark.

ACKNOWLEDGMENTS

I would like to thank my advisor, Associate Professor Michael Bailey, for his guidance and support in this research. I would also like to thank Mika Latimer for her comments, suggestions, insights, and late night discussions that helped move the project forward. I am indebted to Joshua Mason for his mentorship, his feedback, and keeping me focused on moving the work toward completion.

Finally, I would like to express my gratitude to my all my friends who have made the last five years at the University of Illinois memorable.

TABLE OF CONTENTS

LIST OF ABBREVIATIONS

ABI	Application Binary Interface
ATB	ARM Trace Bus
AXI4	Advanced eXtensible Interface 4
CFG	Control Flow Graph
CFI	Control Flow Integrity
CPU	Central Processing Unit
CRA	Code Reuse Attacks
ETM	ARM Embedded Trace Macrocell
FPGA	Field Programmable Gate Array
HSMON	Hardware Signature Monitor
ICF	Indirect Control-Flow
IP core	Semiconductor intellectual-property core
ISA	Instruction Set Architecture
PTM	Program Trace Macrocell
ROP	Return-oriented Programming
RTL	Register Transfer Level circuit description
SoC	System on Chip
TPIU	Trace Port Interface Unit
WNS	Worst Negative Slack
ZynqMP	Xilinx Zynq Ultrascale+

CHAPTER 1

INTRODUCTION

Malware is often constructed from techniques to subvert machine-code execution through reuse of existing program code. In order to bypass data execution prevention (DEP), an exploit typically employs return-oriented or jump-oriented programming. These are known as *code-reuse attacks* (CRA). One way to protect a program from CRAs is to enforce the safety property *control-flow integrity* (CFI). CFI enforces that the execution of a program cannot deviate from its *control-flow graph* (CFG). In this work, we present a CFI policy and hardware design to enforce the policy.

CFI [1] is a defense mechanism that aims to prevent CRAs and all other control-flow hijacking attacks under an attack model where the attacker can read and write any process memory (but not instructions). CFI ensures that the execution of a program follows a path in the CFG. Any control flow transfer that is not described by an edge in the CFG is a CFI violation that terminates the program. Any invalid control flow caused by an attacker will be detected, without regard to the exploit mechanism that caused the invalid target of the transfer.

In this work, we assume the standard attack model for CFI. The operating system and hardware are secure and fault-free. The attacker can change arbitrary data in memory on the stack, heap and global variables of the protected program. Data memory is not executable, and instruction memory is not writable (W$\oplus$X policy). Therefore, the attacker cannot cause the program to execute instructions that the attacker places in memory.

Return-oriented programming (ROP) [2] is a method used by attackers to create new behavior in a program by changing the target of subroutine returns. In ROP, the attacker finds short sequences of instructions in memory that end with return, or another type of indirect control transfer instruction [3]. Several sequences are chained together to achieve arbitrary computation. A perfect CFI policy defeats ROP but it can be desirable to implement a

relaxed CFI policy for improved system performance. A relaxed CFI policy must be careful to ensure that practical exploits cannot be constructed using ROP.

Research on CFI can be broadly categorized as either software enforced or hardware enforced. Software enforcement requires the insertion of new instructions into a program before at the indirect branch instructions and the target sites. Given the source code to a program, this can be done in the compiler. Lacking source code to a program, an approximate CFG may be derived that allows insertion of some CFI checks through binary rewriting. Hardware techniques [4] typically require compiler generated code or binary program rewriting, but they differ from software techniques in that they generally include CFI-specific instructions in the instruction set architecture (ISA). Hardware enforcement allows very low performance overhead, but may require modifications to the processor and the binary program.

The design presented here, Hardware Signature Monitor (HSMON), does not use binary rewriting or code insertion by the compiler. It has low performance overhead, and does not require changes to the processor core or the instruction set. We achieve this using two developments—a CFG path signature technique for enforcing a relaxed CFI and a hardware instruction trace monitor.

Signature modeling [5, 6] is a method used to detect control flow violations for the purpose of fault detection. Each executing instruction updates a signature that is periodically checked against a set of reference values. If there is no corresponding reference value there is a fault. Because we assume error-free hardware, the only control flow violations possible are due to indirect branches. HSMON computes signatures based on only the indirect branch target addresses. An indirect control path signature is a generated from a short sequence of previous indirect control transfer targets. Valid signatures are derived from the CFG and loaded into the hardware just prior to the start of the program. Control flow attacks, including ROP, generate run-time signatures that are not part of the valid set of signatures for a program, so HSMON identifies the control flow violation. The details of signature generation are in Section 3.3.

Modern CPUs generate instruction traces for the purpose of in-system debugging. HSMON uses hardware support for instruction trace to observe control flow paths with a hardware unit that operates asynchronously to the

central processing unit (CPU) core. Instruction traces convey information about the state of a CPU core and the instructions being executed. Example trace data includes branch outcomes, branch targets, exceptions, traps, and cycle counts. Trace data is captured to main memory or output directly to a dedicated trace port. Efficient trace protocols encode only information that cannot be obtained through static code analysis. Variable length trace packets and stateful protocols are used to further reduce bandwidth requirements. Instruction trace monitoring used by HSMON is described in Section 3.2.

Obtaining the CFG by static analysis of a large binary program remains a challenge for current research. Target inference of *indirect control-flow* (ICF) transfers are particularly challenging because unrestricted ICF allows transfer to any target address. One approach, bin-CFI [7], provides a 99% reduction in possible ICF targets. Although HSMON requires static analysis of ICF targets, our research does not address the problem of sophisticated static analysis of ICF targets. Future improvements in this area will enhance the effectiveness of the path signature method used here.

This work is organized as follows. Chapter 2 describes related work. Chapter 3 details the system architecture of the hardware, software and discusses key concepts. Chapter 4 provides a detailed description of the hardware architecture and the microarchitecture of each component. Chapter 5 evaluates the results of the implementation of a prototype system. Chapter 6 concludes with a summary of what we learned and ideas for future work.

CHAPTER 2

BACKGROUND

In this chapter, we first describe a brief history of CRAs and W⊕X. Next, we discuss related work on hardware monitoring IP using instruction traces to detect CRAs.

Traditional stack-smashing attacks consist of injecting executable code, called shellcode, into a program's memory then redirecting the programs control flow to the shellcode. Most commonly, it is the return address that is overwritten by the attacker. W⊕X stops the attacker from writing shellcode into memory. Another method of attack is known as return-to-libc [8]. An attacker uses a library subroutine, typically in the C library, instead of shellcode. The return address is overwritten by the attacker with the address of the library subroutine. This type of attack is not prevented by W⊕X because it does not require injection of shellcode into writable memory. Despite this limitation, W⊕X makes it more difficult for the attacker to exploit vulnerable programs, so it is used widely in commercial operating systems.

Return-oriented programming (ROP) is a generalization of the return-to-libc attack [2, 3]. This technique uses short sequences of instructions, called gadgets, to create new behavior using fragments of the original program code. Each gadget is a short sequence of instructions, in the program, that ends with a return instruction. The attacker writes a sequence of gadget addresses on the stack, so the CPU will execute each gadget in the sequence determined by the attacker. A variation of this is called jump-oriented programming (JOP) [9]. In this case, the attacker finds gadgets that end in a jump (indirect branch) instruction and a special dispatcher gadget. The sequence of gadget addresses is written into memory that the dispatcher gadget reads as a table of addresses where the program should jump. JOP does not need the stack or return instructions, and can be constructed from jump instructions using any CPU register.

To the best of our knowledge, Lee et al. [10] were the first to introduce a

hardware CRA monitor that uses trace monitoring rather than CPU modifications. They implement CRA detection hardware for the 32-bit ARM CoreSight program trace macrocell (PTM). A shadow call stack is used to detect *return-oriented programming* (ROP) attacks. Meta-data gathered by static program analysis is stored in main memory and accessed by the ROP monitor. Performance overhead is reported to be 1 to 4 percent over the baseline but the operating frequency of the host CPU is 200 MHz. Typical configurations of the FPGA device used in that work operate the CPU at 1 GHz.

The next work from Lee et al. [11] uses a similar architecture to the first, with binary program rewriting to replace the main memory meta-data. Detection of *jump-oriented programming* (JOP) is introduced. Performance overhead due to the CRA monitor is reported from 1 to 10 percent. The operating frequency of the CPU is further reduced to 60 MHz in this work. Neither [10, 11] provide sufficient justification for the slow CPU frequency.

Compared to the prior work, our approach uses neither binary rewriting nor main memory. Avoiding both of those means that memory access patterns are not disturbed by the monitor, so we can predict that there will be no performance overhead.

CHAPTER 3

SYSTEM ARCHITECTURE

In this chapter, we give an overview of the system, describe relevant architecture features, key concepts, and the operation of HSMON as it observes the instruction trace of a program.

3.1 System-on-Chip Overview

The prototype system for our design is implemented using a Xilinx Zynq Ultrascale+ EG (ZynqMP). ZynqMP is a heterogeneous system based on a 64-bit ARM quad-core CPU with Xilinx 8-series field programmable gate array (FPGA) on a single 16 nm integrated circuit system-on-chip (SoC) [12]. Four ARM Cortex-A53 cores are the primary component of the processing system. The cores communicates with the programmable logic, or FPGA, through a central interconnect that implements the AMBA AXI4 interface protocol [13].

ZynqMP integrates the ARM CoreSight architecture [14] to provide real-time debug and trace. The CoreSight architecture defines a standard set of interfaces for debug and trace components so that software and hardware from multiple vendors can be integrated on the same SoC. ZynqMP integrates CoreSight component IP cores from ARM to implement the debug and trace architecture [15, 16]. CoreSight components form the communication path that connects HSMON to the ARM Cortex-A53 trace unit.

The ARM embedded trace macrocell (ETM) is a component of each CPU core in the processing system of the ZynqMP. It gathers information from the processor pipeline and encodes that in a format called the ETMv4 trace protocol. Encoded instruction trace packets are placed on the ARM trace bus (ATB) of the processing system. The programmable logic reads from the ATB through the trace port interface unit (TPIU).

HSMON is designed in the hardware description language SystemVerilog, synthesized, placed, and routed in the programmable logic. It connects to the central interconnect with an AXI4 slave port and receives trace data from the TPIU. Figure 3.1 depicts the architecture of the system.

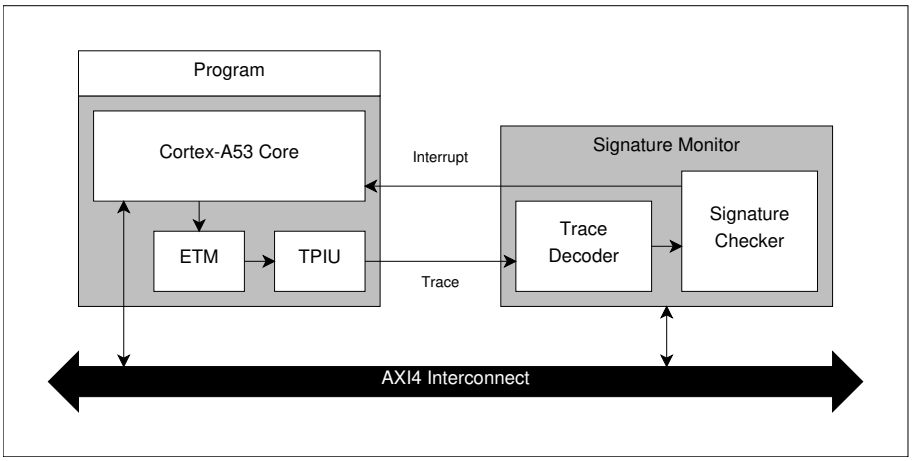

Figure 3.1: System architecture.

The ETM and the TPIU are built-in hardware modules of the processing system. Both are activated by a driver program that accesses each devices memory-mapped registers. In this system, we configure the ETM to emit only instruction trace packets for user-mode programs. No trace information for supervisor (or hypervisor) modes are sent to the TPIU. The TPIU is configured to format the trace stream with trace source identifiers so multiple copies of HSMON may be synthesized in the same programmable logic, one for each CPU core. The trace decoder front end of HSMON is configured through memory-mapped input/output IO) with the trace source identifier of the CPU core that is to be monitored. Instruction trace packets are of variable length. The trace decoder tags the byte that ends each packet with an additional bit to indicate which byte ends a packet. From there, the packets are sent to the signature checker. Further detail on the hardware architecture of the trace decoder and signature checker are in Chapter 4.

3.2 Trace Monitoring

The ETMv4 [17] instruction trace protocol is designed to minimize trace bandwidth. Conventional usage of the trace port for system debugging would

connect a protocol analyzer to the trace port pins external to the integrated circuit package. Due to the bandwidth constraints, the architects at ARM chose to only emit trace information that cannot be inferred by static analysis. Most importantly, direct branch instruction trace information is emitted in a compressed form called *atoms*. Atom packets indicate only the branch outcome, that is, branch taken or branch not taken. Straight through code, when the program counter increments to the subsequent instruction, does not emit any packets. Several encodings of address packets indicate an indirect control flow transfer to a target that cannot be inferred through static analysis. Instructions that emit address packets are listed in Table 3.1 [18]. Context packets indicate the hardware context ID, which is set to the process ID by the Linux kernel during an OS context switch. A context packet also indicates the processors exception level, virtual machine identifier, and 64-bit or 32-bit instruction mode.

Table 3.1: Frequently used address packet generating instructions.

Mnemonic	Instruction
`BLR`	Branch with link to register
`BR`	Branch to register
`RET`	Return from subroutine
`BRK`	Breakpoint
`HLT`	Halt
`HVC`	Generate exception to hypervisor
`SMC`	Generate exception to secure monitor
`SVC`	Generate exception to operating system
`ERET`	Exception return

Enforcing a strong CFI policy requires us to validate all indirect control flow transfers. The trace protocol we have just discussed does not indicate the source of an indirect control transfer—only the target. In order to know the source and validate possible targets we would need to store the CFG in a manner accessible to the trace monitor hardware so that it can follow the CFG using the atom packets until an address packet is observed. At that point the monitor would know the source of the transfer.

We avoid storing the complete CFG by analysis of only paths of indirect control flow to determine the targets. This method allows us to use only in-

formation present in the trace protocol, with relatively little storage required compared to storing the entire CFG.

3.3 Indirect Control Path Signatures

As discussed in the previous section, direct control flow transfers are only visible in the instruction trace as atom packets that indicate whether a branch is taken or not taken. Indirect control transfers caused by *RET*, *BR*, and *BLR* instructions encode the target address in the instruction trace. Exception, interrupts and system calls encode the target address and the exception return address in the instruction trace at the time the exception, interrupt or system call occurs.

In order to reduce the storage requirement of information gathered through static analysis, we introduce indirect control path signatures. An indirect control path signature is a hash computed from the target address of each indirect control transfer instruction. We compare the computed signature against each reference signature when HSMON detects a context switch caused by the *SVC* instruction; when the monitor detects a system call. Static analysis of the CFG derives the set of indirect control path signatures by backwards traversal of the CFG along all paths leading to a basic block containing a *SVC* instruction. Due to the presense of loops in most programs, all paths traversal of the CFG may never terminate and the set of hashes may be of infinite size. To avoid this problem we fix the number of indirect branch addresses that make up the signature to a small number.

Figure 3.2 shows a small program with several indirect branches that we will use to understand how the signature is generated. In this example, we limit the number of address hashes that make up a signature to $N = 4$. In order to generate the set of signatures, we must start at the instruction following each *SVC* instruction and traverse the CFG backwards to reach either address hash limit N or the "_start" symbol. The instruction trace emits the exception return address when a system call is taken, so we will use that as the first address hash $S_1 = H(d)$ in the signature. Moving backward, we see that block **b** is a return target because it follows a *BLR* instruction, so the address hash becomes $S_1 = H(d) \oplus H(b)$. Block (g) is the target of a system call return so it is also part of the signature. Now we have

```
.text
.globl _start
_start:     adr x1, h
            ldr x20, j
            blr x20
a:          adr x1, i
            ldr x20, k
            blr x20
b:          mov x0, 0
            mov x8, 93
c:          svc 0
d:          nop

e:          mov x0, 0
            mov x2, 6
            mov x8, 64
f:          svc 0
g:          ret
.data
h:          .ascii "hello   "
i:          .ascii "arm  \n  "
j: .quad e
k: .quad e
```

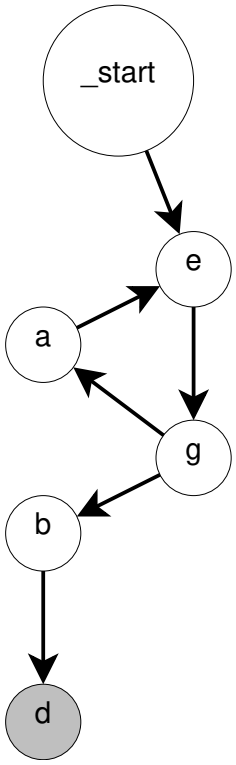

Figure 3.2: Example assembly language source code and CFG.

$S_1 = H(d) \oplus H(b) \oplus H(g)$. Finally, we reach block **e**, which terminates the algorthim because it is the fourth address to get

$$S_1 = H(d) \oplus H(b) \oplus H(g) \oplus H(e)$$

The second system call returns to block **g** so we start with $S_2 = H(g)$, then walk backward to get $S_2 = H(g) \oplus H(e)$. Block **e** is the target of two indirect branch and link instructions. We will follow both forks to compute two signatures, S_2 and S_3. Choosing block **a** we see that it is a return target because it follows *BLR*, so we get $S_2 = H(g) \oplus H(e) \oplus H(a)$. As previously shown, we include block **g** because it is the target of a system call return, so the final signature is

$$S_2 = H(g) \oplus H(e) \oplus H(a) \oplus H(g)$$

The third signature is generated from where we forked S_2 at call target **e**. There we had $S_3 = H(g) \oplus H(e)$. The global "_start" symbol is always

an indirect target on our prototype platform, so we include it. The final signature is

$$S_3 = H(g) \oplus H(e) \oplus H(\_\text{start})$$

Now that we understand how to generate a set of signatures through static analysis let us explore how a signature is computed when a program runs on the CPU core (forward CFG traversal). The set of signatures is repeated below for your reference while reading Section 3.4.

$$\begin{aligned} S_1 &= H(d) \oplus H(b) \oplus H(g) \oplus H(e) \\ S_2 &= H(g) \oplus H(e) \oplus H(a) \oplus H(g) \\ S_3 &= H(g) \oplus H(e) \oplus H(\_\text{start}) \end{aligned}$$

3.4 Run-Time Signature Monitoring

Generating the signature set is only necessary once for a given program binary. Either an external tool or the operating system can analyze the program to derive the allowed signatures. We must also generate signatures during execution of the program to check against the allowed signatures. We will understand how this works by example using Figure 3.2 from Section 3.3. For this, we focus our attention on the assembly listing on the left side of the figure.

We start with an initial run-time signature initialized to $S_R = 0$. The program begins with an indirect jump to the "_start" symbol so the signature becomes $S_R = H(\_\text{start})$. It proceeds to the end of the block and follows the indirect branch to instruction **e**, so we update the signature to $S_R = H(\_\text{start}) \oplus H(e)$. Now the program reaches the system call at **f**, so we include the system call return address in the signature and it becomes $S_R = H(\_\text{start}) \oplus H(e) \oplus H(g)$. Because the exclusive-or (XOR) operator is commutative and associative, it is easy to show that $S_R = S_3$, so the run-time signature is equal to the signature derived through static analysis.

The program continues after the system call return. The instruction **g** returns to **a** so we update the signature to $S_R = H(\_\text{start}) \oplus H(e) \oplus H(g) \oplus H(a)$. The next indirect branch transfers control to **e** and we have reached

the limit of $N = 4$ hashes in the signature. We remove the oldest hash and add the new one, $S_R = H(e) \oplus H(g) \oplus H(a) \oplus H(e)$. The system call is reached so the signature is now $S_R = H(g) \oplus H(a) \oplus H(e) \oplus H(g)$. Now $S_R = S_2$ so we have a valid signature according to our previous static analysis.

After the system call returns, the instruction **g** returns to **b** and the signature becomes $S_R = H(a) \oplus H(e) \oplus H(g) \oplus H(b)$. The next control transfer occurs at **c**. Even though this system call cannot return, we include the system call return **d** as we have with other system calls. The signature is now $S_R = H(e) \oplus H(g) \oplus H(b) \oplus H(d)$. Just like before, the signature matches on the of the static analysis signatures. $S_R = S_1$.

As shown by the previous example, an important consideration is that the operator used to compute the signature from the address hashes needs to have the communitativity property and the associativity property. Because we initialized the S_R to 0, we also need the identity $A \oplus 0 = A$ for any A. Another property of XOR is that each value is its own inverse. For each A, $A \oplus A = 0$. In the previous example, we did not need this property. It is useful for efficient implementation of run-time signature monitoring for large values of N. Instead of using XOR $N - 1$ times every time the signature is checked, we use it exactly three times when the signature is updated. Consider one of the updates in the prior example. The signature is $S_R = H(\_\text{start}) \oplus H(e) \oplus H(g) \oplus H(a)$ and the program transfers to **e**. Assume the value of S_R has already been computed. We find the next value S'_R by computing the expression

$$S'_R = S_R \oplus H(e) \oplus H(\_\text{start})$$

Other operations can be used if they have the same properties. We use bitwise XOR because it can be computed with fast circuits in hardware for any fixed bit width output hash function.

We define a *violation* in this system as a run-time signature that is not found in the set of signatures derived through static analysis. In the code example we have discussed so far it is possible for an attacker to cause a violation under our attack model. The data section of the program is controlled by the attacker. Code pointers are stored at locations labeled **j** and **k** in data memory.

No library code is used in the example shown in Figure 3.2. For this dis-

cussion, let us assume a modified program that has additional executable code in memory that is not reachable during normal (no-attack) code execution (dead code). Now suppose that an attacker alters code pointer **k** to point to instructions in the dead code prior to the execution of block **a**. Any *SVC* instruction in the dead code will cause a violation because the address following the *SVC* instruction is part of the signature. The attacker must reuse the existing system call instructions that are part of the normal code. It is reasonable to assume that no dead code would have a direct control transfer to any label other than **e** because it is the only subroutine in the example. Let us consider the case where the attacker attempts to use the system call at **f** by using a direct control transfer to **e**. The signature at the system call is $H(g) \oplus H(a) \oplus H(\text{deadcode}) \oplus H(g)$. Now consider the case where the attacker cause an indirect control transfer to **e**. The signature is $H(a) \oplus H(\text{deadcode}) \oplus H(e) \oplus H(g)$.

A gadget could exist in the dead code that allows the attacker to cause an indirect transfer to the same address *deadcode* a second time before an indirect transfer to **e**. We must consider what happens in this case. The signature on the first control transfer to *deadcode* is $H(g) \oplus H(a) \oplus H(e) \oplus H(\text{deadcode})$. The second control transfer updates it to $H(a) \oplus H(e)$ because of the properties of XOR discussed previously. It is easy to see that neither an indirect nor direct transfer to **e** will generate a valid signature, so a violation will occur.

Now we understand the methods of generating signatures from static analysis and run-time instruction execution. We still need a method to store the set of allowed signatures that facilitates fast access. In the next section we describe how we store a large set of signatures, test set membership quickly, and discuss the limitations of our approach.

3.5 Signature Storage

The final concept we explore before describing the operation of HSMON is the the data structure and algorithm for storing the signatures that are derived by the static analyis. We want this storage to be small in size, offer low latency read access, high bandwidth write access for transferring the signature set into the monitor device, and be simple to implement in

hardware.

We use the Bloom filter [19] data structure to satisfy these requirements. A Bloom filter is a probablistic data structure used to test if an element is a member of as set. A Bloom filter allows errors in order to be more time and space efficient than algorithms that test set membership without errors. Only false positives are possible. A false positive indicates that an element is a member of the set, when it is not. False negatives do not occur. In our case, a false positive allows a signature that should be a violation.

We consider this to be an acceptable tradeoff for increased performance of the real-time monitor. As long as the error rate low, it is difficult for an attacker to find a code path that causes a false positive. The primary impediment is that the attacker does not know the seed input to the hash function H. The attacker has read access to all memory in the process under our attack model but the signature set and the seed input to the hash function is not part of the process memory. An attacker would not be able to compute the set of signatures without the hash function seed values.

3.6 HSMON Operation

In this section we describe the operation of HSMON in a system running the Linux operating system. HSMON continuously monitors the instruction trace received through the trace port. It operates in parallel and asynchronous to the CPU core that it monitors. The hardware monitor runs passively until enabled. Only two event counters operate when it is disabled. One counts the number of 4-byte trace words received from the selected CPU core. The other counts the number of 4-byte trace words that are lost (or dropped) by the monitor.

Prior to being enabled, the monitor is programmed with the Context ID of the program of interest. In Linux, the context ID is the same as the operating system process ID. The monitor is also programmed with the start address from which to being actively monitoring the trace stream. This is usually the "_start" or "main" symbol of the program. It is also programmed with the Bloom filter bits that encode the signature set. An additional Bloom filter memory records a log of all signatures observed since the start address, but the memory must be cleared manually before the program starts.

Once programmed, the monitor must be enabled before the program begins execution. The monitor ignores all events until the programmed start address is observed in the trace stream. Beginning with this address, the indirect control flow path signature is updated on every observed target address and system call in user mode for the programmed context ID. Once a system call is observed in the trace stream, the monitor checks the Bloom filter to know if the computed signature is in the set. If it is not, the violation counter increments and an interrupt is raised.

Each system call increments the system call counter. At the same time, the bits generated from the signature that are checked against the Bloom filter are recorded in the the log memory. In some cases, the log memory may be used to build the set of signatures over time by repeatedly running the program under different inputs. The log memory is in the same format as the Bloom filter data so it can be saved and reloaded later.

Further details of the operation of the hardware monitor can be found in Chapter 4, which describes the design of the hardware and software that controls HSMON. Detailed programming information is described, and a register reference appears at the end of this document as Appendix A.

CHAPTER 4

HARDWARE DESIGN

HSMON is a semiconductor intellectual property core (IP core) that is integrated into SoC designs that use ARM CoreSight components. HSMON receives data from the TPIU component. It is configured by the operating system software through an AXI4-Lite memory-mapped slave port. In this chapter, we describe the design and operational details of each component of the HSMON IP core.

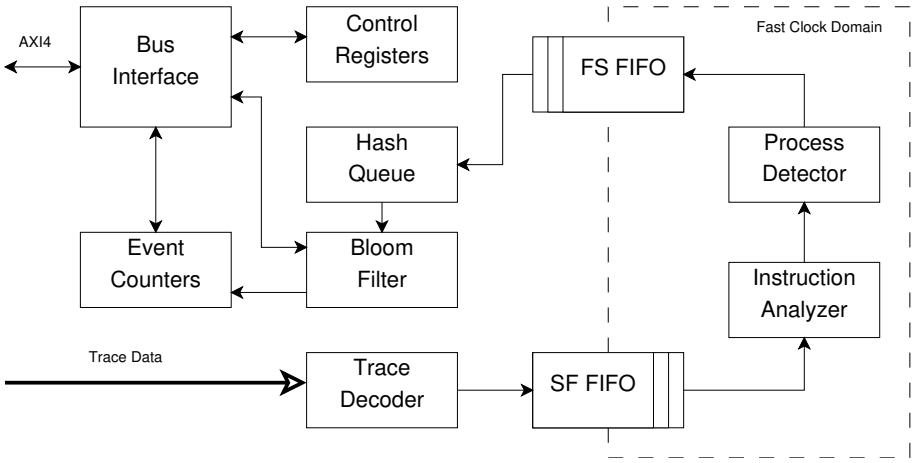

Figure 4.1: HSMON block diagram.

Figure 4.1 shows the main blocks of HSMON. HSMON is divided into two clock domains, the slow clock domain and fast clock domain. The slow clock domain contains the bus and trace port interface logic, as well as the hash queue and Bloom filter. The fast clock domain contains the instruction analyzer and process detector.

The purpose of each component is as follows:

Bus Interface AXI4-Lite slave bus interface for programming HSMON.

Event Counters Event counters for system calls, exceptions, system call violations, SF FIFO dropped inputs, and 32-bit trace words.

Trace Decoder Decodes TPIU frames, filters trace streams, and flags the end byte of each instruction trace packet.

SF FIFO Transports 32-bit wide trace data in the slow clock domain to 8-bit data in the fast clock domain.

Instruction Analyzer Decodes address and context switch packets from the instruction trace stream.

Process Detector Monitors address and context to identify packets corresponding to the program of interest.

FS FIFO Transports 64-bit address and status bits from the fast clock domain to the slow clock domain.

Hash Queue Computes the hash of the incoming address and stores the hashes of the prior N addresses. Also computes the XOR of all queued hashes.

Bloom Filter Checks the XOR of the address hash queue against the Bloom filter data uploaded through the programming interface.

4.1 Microarchitecture

The microarchitecture of selected components is described in this section. This is not intended to describe the detailed operation of every component. More detail is provided when the operation of a module is non-trivial.

4.1.1 Trace Decoder

The Trace Decoder is a pipeline that consists of five modules. Each module accepts the output of the previous module in the pipeline. The pipeline begins with the TPIU output data and ends with the ETM instruction trace bytes, augmented by one bit per byte to indicate the bytes that finish a packet. Figure 4.2 illustrates the trace decoder pipeline.

The sync remove module detects the 32-bit full-sync word that is periodically output from the TPIU just prior to a 16 byte frame. After reset, it discards all incoming data until the full-sync word is detected. Once the

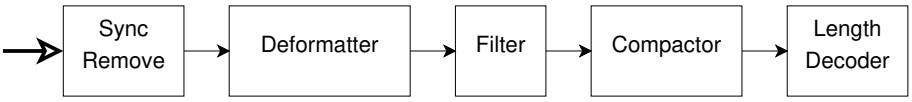

Figure 4.2: Trace decoder block diagram.

first full-sync word is detected, input is passed to deformatter with only the full-sync words omitted.

The deformatter decodes the 16-byte frame format output by the TPIU. The microarchitecture is a five-stage pipeline so that a full frame is buffered in the module. Two output ports decode data for up to two trace IDs for each input word. Both output ports pass data to the filter module.

The filter accepts up to four bytes on input port A and up to two bytes on input port B per cycle with a distinct trace ID per port. A 7-bit trace ID port indicates which trace ID will be passed through to the output, all others are discarded. A single output port provides data to the compactor.

The compactor receives up to four bytes per cycle and outputs exactly four bytes on each cycle that it produces output. A single four byte internal register holds the input until four bytes can be output. Assuming a full four byte input, data is moved to the length decoder with a one cycle delay.

The length decoder is the most sophisticated module in the trace decoder. Figure 4.3 shows a diagram of the microarchitecture. It can be understood as a sequence detector FSM for eight bit inputs. Instead of the usual arrangement of next-state (NS) logic and storage element, the NS logic is duplicated four times so that four input bytes are processed every cycle. The output of the length decoder is four data bytes and four bits per cycle. Each of the four additional bits indicates whether the corresponding byte is the end of an instruction trace packet. The output is moved to the fast domain through an asynchronous first-in, first-out queue (FIFO) connected to the instruction analyzer.

4.1.2 Instruction Analyzer

The instruction analyzer decodes each address and context packet in the instruction trace data stream. It retains and updates internal state information needed for correct decoding of future packets. The output is the fully

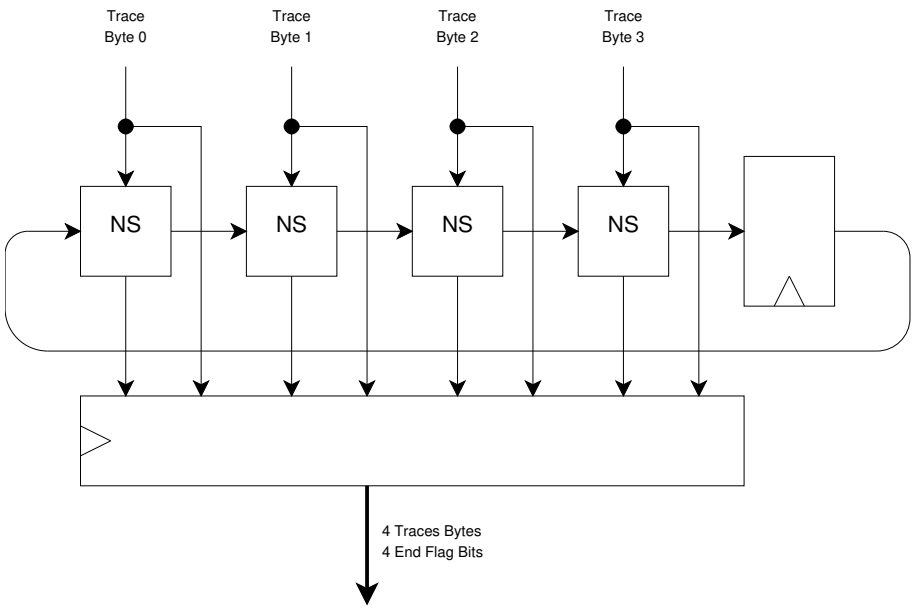

Figure 4.3: Length decoder microarchitecture.

decoded instruction address, exception type, and context ID. Three valid signals indicate which of the three output types were decoded, even in the case where the value did not change. One additional signal indicates that the internal state was reset. The instruction analyzer data path is shown in Figure 4.4. Combinational logic (not shown) controls the register load signals and multiplexers.

The trace buffer captures sequential packet bytes from the incoming trace data in an 18-byte wide register to allow parallel access to the complete packet. It generates a valid signal to indicate to the control unit that a complete packet is present on the wide output. Additionally, a sequential output is provided with start byte and end byte signal. Note that the trace decoder output only provides the end byte signal and the start byte signal is generated internal to this module.

The packet detector and address detector modules that identify packet headers and address formats. The packet detector identifies the nine packet types that cause updates to the internal states from the sequential output of the trace buffer. It outputs the packet type to the control unit. The address detector identifies the address format used by the packet from the output of the address multiplexer. The address multiplexer selects from the three positions that an address packet can occur within the wide packet output of

the trace buffer.

Timestamp decoder uses the wide output from the trace buffer to decode updates to the timestamp register. The address decoder applies updates to the address shift register chain based on the output of the address multiplexer. The context multiplexer selects from the seven positions where a context update may occur within the wide packet output of the trace buffer. From there, the context and virtual machine ID (VMID) decoder decodes the context ID, context flags, and VMID as appropriate for the packet encoding.

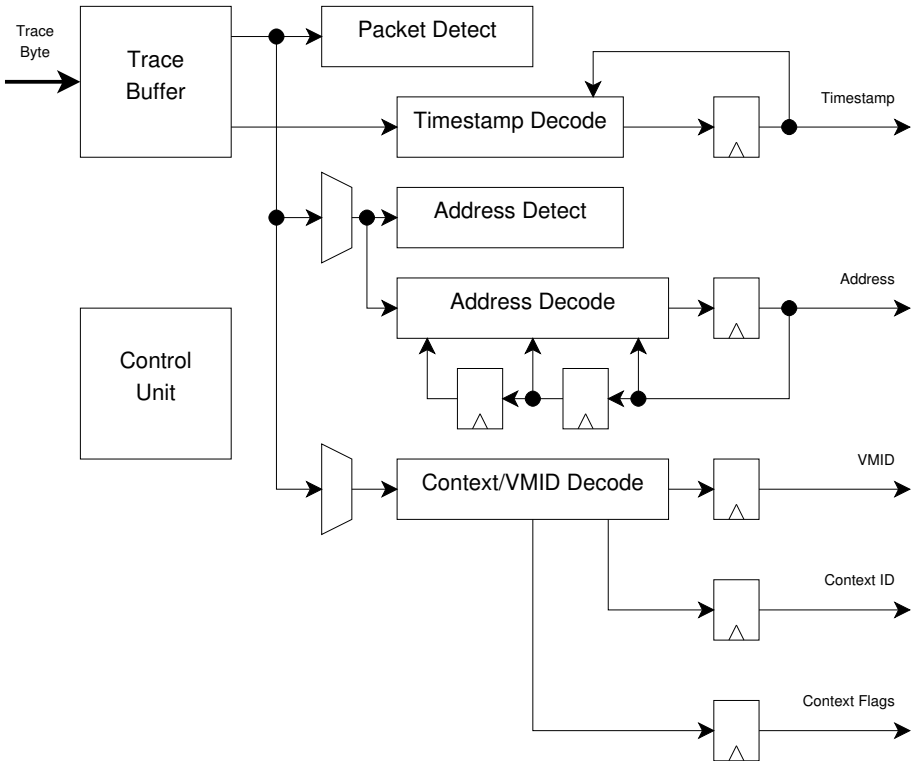

Figure 4.4: Instruction analyzer microarchitecture.

4.1.3 Hash Function

The hash function module implements the widely used MurmurHash3 function. Figure 4.5 shows the 12-stage arithmatic pipeline. It accepts a single 64-bit input and 32-bit seed value each cycle. It outputs a 32-bit hashed value every cycle, with a 12-cycle latency from input to output.

This processing pipeline achieves much higher throughput than required. It can hash a new input every cycle, but address packets only update at most

every 8 cycles. An improved design would reduce the number of function units by sequencing the input over multiple cycles.

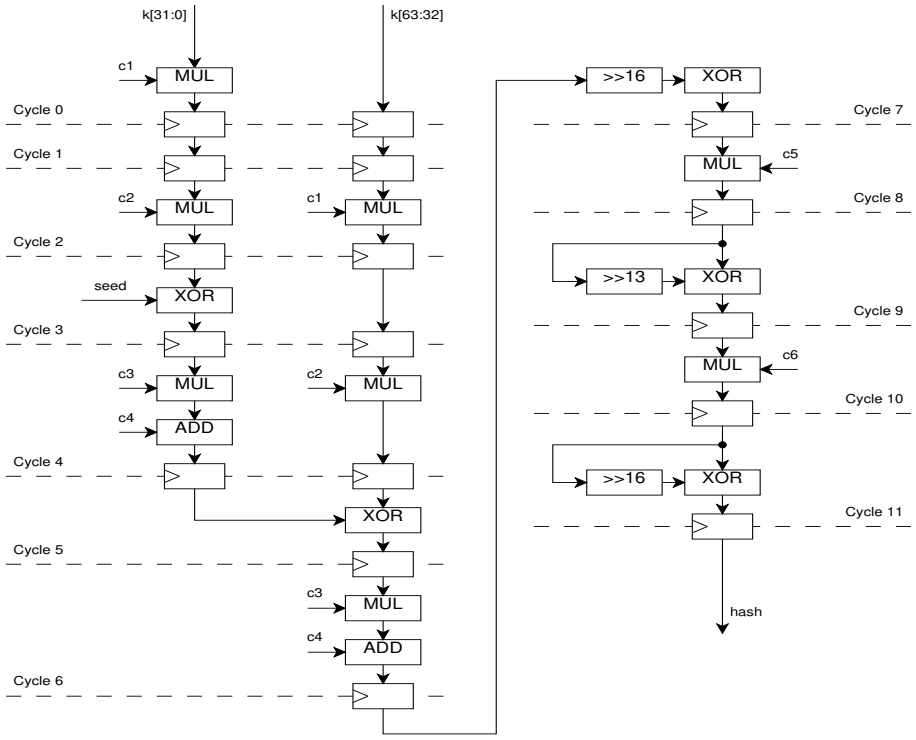

Figure 4.5: MurmurHash3 function pipeline.

4.1.4 Hash Queue

The hash queue holds the last N hashed address values generated from the trace stream. N is a fixed constant that is set before hardware synthesis. The module's output is the XOR of the previous N hashed address values. A shift register tracks the previous N hashes and the output is computed by the XOR of the oldest hash, newest hash, and the previous output hash. Figure 4.6 shows the microarchitecture of the hash queue. Two hash queues are used to generate a 64-bit hash input to the Bloom filter module.

4.1.5 Bloom Filter

The Bloom filter module computes a 65536 bit pattern containing one to four 1-bits from the output of the hash queue. The 64-bit input hash is split into

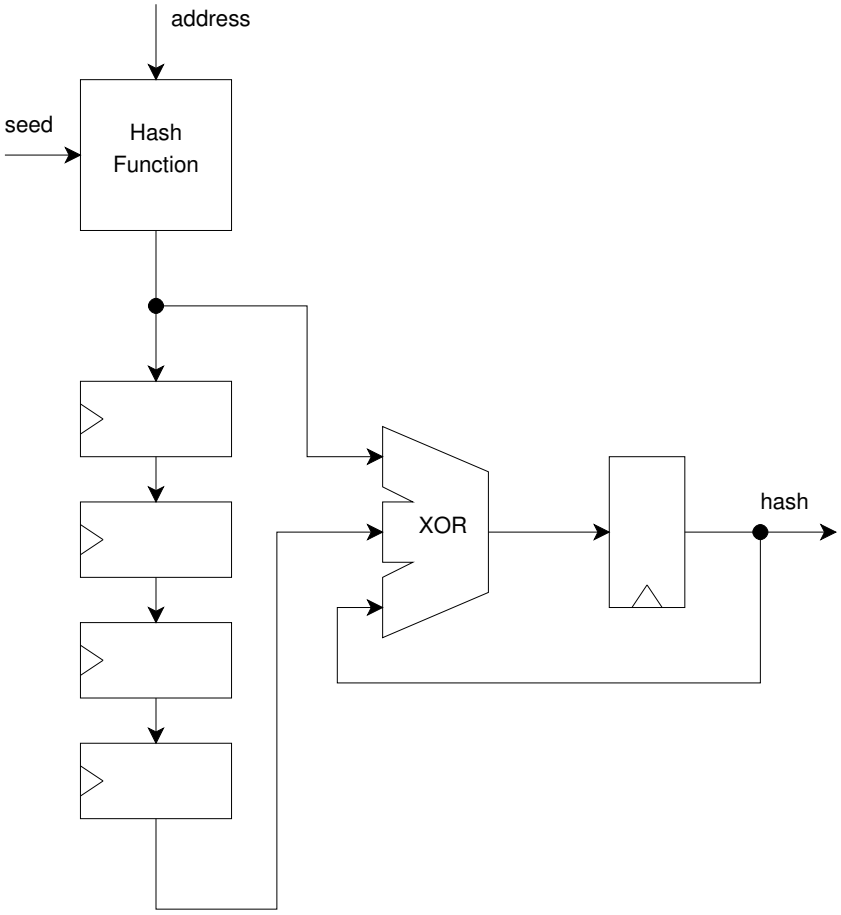

Figure 4.6: Hash queue microarchitecture.

four 16-bit components. A 1-bit in the pattern is computed by the function 2^k where k is the 16-bit component of the input hash. The Bloom filter output is 1 when each 1-bit in the pattern is also a 1-bit in the Bloom filter data. Otherwise, the output is 0.

Figure 4.7 shows the microarchitecture of the Bloom filter. The bus interface to read/write the Bloom filter memory and the Bloom filter log memory is not shown. The Bloom filter data is stored in a 2048 by 32 bit block RAM. Instead of computing 2^k directly, the microarchitecture first computes $\lfloor \frac{k}{32} \rfloor$ to find the address then computes $2^{(k \mod 32)}$ to compare with the data read from memory. The Bloom filter data is loaded through the bus interface into the memory array. A Bloom filter check begins when a new hash input is received and the *check* input is 1.

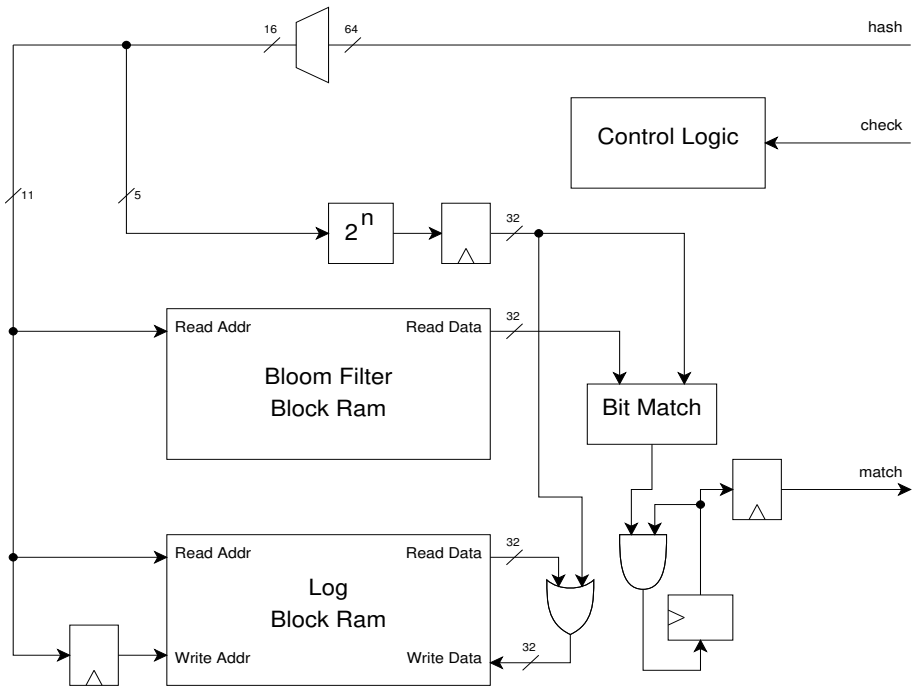

Figure 4.7: Bloom filter microarchitecture.

4.1.6 Event Counters

The final important component of the design is a set of event counters that allow system software to measure the number of failed matches in the Bloom Filter, number of system calls, exceptions, and trace words observed by HSMON.

4.2 Programmer's Interface

HSMON is software programmable through the AXI4-Lite memory-mapped interface. Each register must be written with a 32-bit aligned data transfer. Table A.1 shows the register set available to the programmer. All registers are initialized to zero on circuit reset. A write access to any register aside from the control register works in the conventional manner. A write access to the control register toggles each bit in the register where the corresponding location equals 1 in the write data. Bits that are 0 in the write data are not toggled in the register. For example, writing the decimal value 1 sets the ENABLE bit to 1 if it was previously 0, and 0 if it was previously 1. Table A.2 describes each bit in the control register.

Control bits enable optional features of HSMON and allow software triggered reset of HSMON hardware. The RESET control bit clears registers mapped above 128 when set. It also resets internal registers in the fast clock domain, the hash queue, the Bloom filter, event counters, and LOCK bit. It does not clear the Bloom filter internal memory. The LOCK control bit disables memory mapped access to registers numbered above 128. The LOCK bit cannot be toggled off using a write to the bit. This ensures that once the seed values are programmed they are no longer readable. The READONCE control bit enables atomic counter reset during a memory-mapped read access to the counter. This allows software to access counters periodically without the need for synchronization between HSMON and the software. Mapping between the hash queue units and the Bloom filter bit positions is controlled by the bits TWOHASH and TWOBIT. The use of these bits to control mapping between signatures and Bloom filter bit patterns is described later in this section. The FERESET control bit clears registers internal to the trace decoder. When the filter ID is changed the finite state machine in the trace decoder could become stuck in an error state. Toggling FERESET will restart the trace decoder. A single interrupt and a set of event counters provide feedback to the system software. The interrupt is set when a violation occurs and clear when software clears the INTERRUPT bit in the the control registers is cleared.

The ENABLE control bit restarts the process detector when the bit flips from 0 to 1. The process detector watches the trace stream until the context ID and start address are observed. Context ID and start address registers should not change when the ENABLE bit is 1. The register outputs cross clock domains without synchronization hardware, so a transient timing fault could occur due to a flip-flop metastable state. Clearing the ENABLE bit stops the process detector. No interrupts will occur and counters will not increment.

There are five event counters. Each counter has a width of 32-bits and an overflow flag readable from the status register. An overflow flag is set when the counter increments from a non-zero value to zero and cleared by software reset. The first three counters indicate events that occur in the trace stream when the CPU core is in user mode and the CPU context ID matches the HSMON context ID. Syscall count increments when a system call is detected. Exception count increments when an exception is observed

in user mode. An example is an access to an unmapped page in virtual memory that is then mapped by the operating system exception handler. Violation count increments when a signature violation is detected. The last two events do not correspond to the process selected by the Process Detector. Drop count increments when the SF FIFO is full. All count increments once for each 32-bit word written to the SF FIFO. Both counters increment even when ENABLE is 0.

In order to understand how to program the set of signatures, we must first understand how a signature maps to bits in the Bloom filter and how each bit position maps to a 32-bit word in the Bloom filter memory. Four possible mappings are selected using the TWOHASH and TWOBIT control bits. When TWOHASH is set to 1, a second 32-bit signature is generated using the Seed2 register to set the hash seed. In the following discussion, S_1 is the first 32-bit signature generated using the value in the seed register and S_2 is generated using the seed2 register value. The two signatures are mapped to a set of two or four 16-bit values. Each indicates a bit position that must be 1 in the 65536-bit Bloom filter memory. Because the memory is 32-bit word addressable through the memory mapped interface, we must first compute the word address A and the bit position within the 32-bit word B. We take "bit slices" of the 16-bit values Z_i to compute

$$\begin{aligned} A_i &= Z_i\,[15:5] \\ B_i &= Z_i\,[4:0] \end{aligned}$$

Bit position B must be set at address A in the Bloom filter memory for this 16-bit value.

The mapping of the two signatures S_1 and S_2 are as follows. If both control bits, TWOHASH and TWOBIT, are 0 the mapping is

$$\begin{aligned} Z_1 &= S_1\,[7:0] \\ Z_2 &= S_1\,[15:8] \\ Z_3 &= S_1\,[23:16] \\ Z_4 &= S_1\,[31:24] \end{aligned}$$

The upper 8 bits of each Z_i are 0. This mapping effectively reduces the Bloom filter from 65536-bits to 256-bits and should only be used for testing.

If only TWOHASH is 1, the mapping is

$$\begin{aligned} Z_1 &= S_1[15:0] \\ Z_2 &= S_1[31:16] \\ Z_3 &= S_2[15:0] \\ Z_4 &= S_2[31:16] \end{aligned}$$

If only TWOBIT is 1 only two 16-bit values are mapped as

$$\begin{aligned} Z_1 &= S_1[15:0] \\ Z_2 &= S_1[31:16] \end{aligned}$$

If both TWOHASH and TWOBIT are 1 only two 16-bit values are mapped as

$$\begin{aligned} Z_1 &= S_1[15:0] \\ Z_2 &= S_2[15:0] \end{aligned}$$

We expect most programs will use the second mapping. The first is suitable for testing and the third minimizes the run time of the static analysis process.

The data in the Bloom filter memory is programmed using a pair of memory-mapped address and data registers. Bloom filter Address is a register that contains an 11-bit pointer to a location in the Bloom filter memory. Reading the Bloom filter data register returns the data from the memory location addressed by the pointer. The pointer is incremented after the read. Similarly, writing the Bloom filter data register writes to the location addressed by the pointer and the pointer is incremented after the write. Bloom filter tested data accesses the memory where a log where bits that are tested in the Bloom filter are recorded. The same address pointer is used for both memories. Neither memory is automatically cleared by any control bit.

The Bloom filter log memory can be used to detect errors in static analysis. For example, the log memory can be read from HSMON after the program exits. Compare the log memory and the Bloom filter data at each bit position. Any single bit value of 1 in the log memory that has a value of 0 at the same position in the Bloom filter data is the due to an observed run-time signature that was not determined through static analysis. Assuming that an attacker

has not altered the control flow of the program, this indicates an error in static analysis. The log memory can be saved and reloaded into the data memory. It can be done repeatedly to build up a set of allowed signatures by exercising the program under various inputs. This is not recommended for normal use because it introduces the posibility of false negatives, or safe control flow paths that are detected as violations.

4.3 Software Support and Tools

In order for HSMON to identify a running process, the OS kernel must be configured at build time to set the process ID in the CPU context ID register. No OS kernel code changes are required for HSMON to operate as an event counter. Interrupt support requires a small driver so processes can be automatically killed due to a violation.

Two software programs are provided for using HSMON on the Linux operating system. Both are command line programs that are run from the shell. The first, Tracer, configures and enables CoreSight components to send trace data to the TPIU. The second, SMTool, controls the HSMON hardware through the memory-mapped bus interface.

SMTool has three capabilities. First, it provides direct access to the read and write memory-mapped registers in HSMON from the command shell. Second, it saves and loads register values from binary files and can print a summary output of the register states. Last, and most important, it can program the registers and execute a user-specified command under the watch of HSMON. Figure 4.8 shows the output of SMTool running a program that invokes three system calls. In this mode, SMTool invokes the program listed at the end of its command line options as a child process. It looks for a file next to the program binary with the suffix ".hwdat" and reads the HSMON configuration from that file. The file format is documented in Appendix C. Prior to invoking the program, SMTool resets HSMON and programs the registers and Bloom filter memory with the values from the filesystem. Appendix B lists the sequence that is used. SMTool does not set the filter ID and always clears the Bloom filter log. A Unix manual page documenting the usage SMTool is in Appendix D.

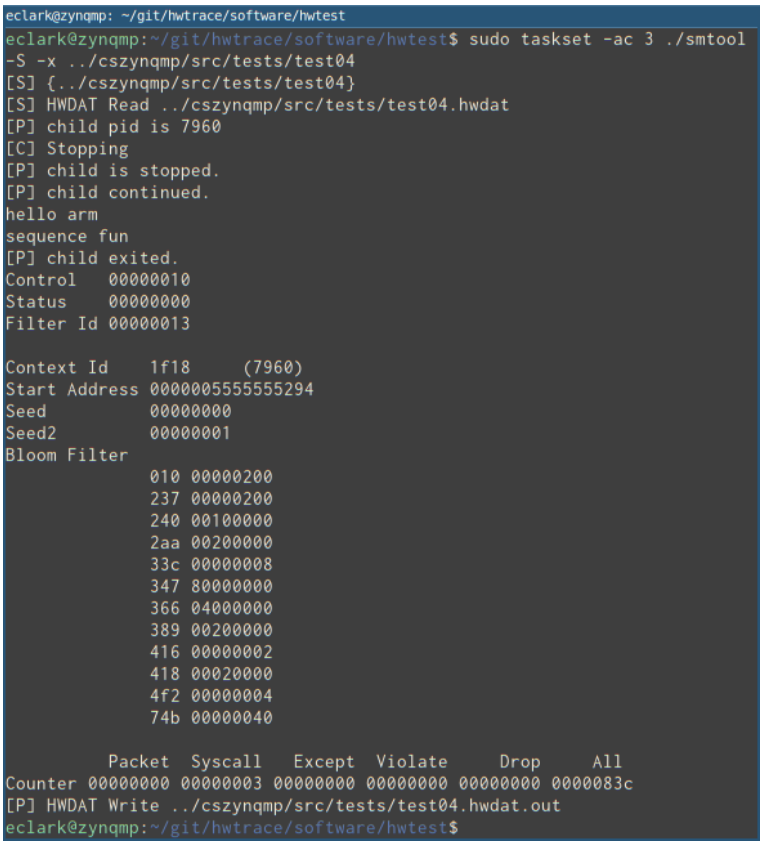

Figure 4.8: SMTool running a small program monitored by HSMON.

CHAPTER 5

RESULTS

5.1 Implementation

In this chapter we present our findings after implementation of HSMON in the prototype system based on the Zynq Ultrascale+ device. We present and discuss the results of implementation of the HSMON design in the FPGA fabric with a focus on timing, area, and power.

The hardware design described in Chapter 4 was entered in SystemVerilog and implemented with Xilinx Vivado 2017.2. It is synthesized for the Xilinx series 8 architecture programmable logic in the XCZU2EG-SFVC784-1-E.

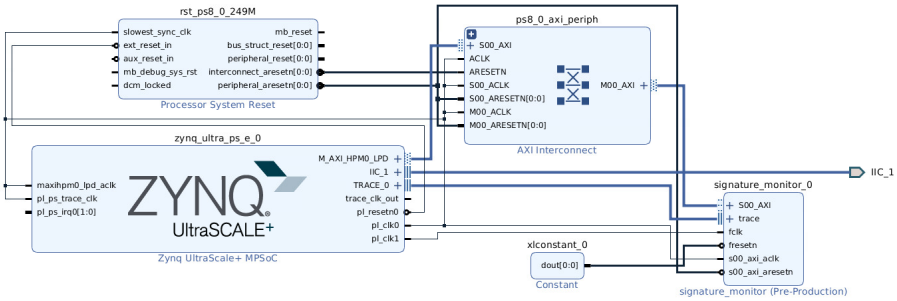

Figure 5.1: Vivado IP integrator block diagram.

Figure 5.1 shows the HSMON IP core in the Vivado IP Integrator tool for our prototype system. The clock frequencies of each part of the system are listed in Table 5.1. The clock ACPU is set to the maximum allowed frequency for the Xilinx Zynq Ultrascale+. The DDR memory clock is set to the maximum frequency recommended by the circuit board designer.

Clock frequencies for the HSMON IP core were set conservatively to ease timing closure. All FPGA resources were automatically placed and routed. The critical paths through HSMON in the slow clock domain part of the length decoder are shown in Figure 4.3. The next state (NS) modules consist

Table 5.1: Clock frequencies in the prototype system.

Clock	Freq. (MHz)	Description
PL0	200	Programmable logic clock 0 is used by the TPIU, AXI4 interconnect and the HSMON slow clock domain.
PL1	400	Programmable logic clock 1 is used for the HSMON fast clock domain.
ACPU	1200	Cortex-A53 CPU clock.
DDR	600	Processing system main memory clock.

only of combinational logic paths, so there is a long combinational path between the outputs and inputs of the state registers. This is necessary to allow the length decoder to decode four bytes per cycle with a variable length encoding. We anticpated that this long path would meet timing at 250 MHz, the maximum frequency of the TPIU that supplies data to HSMON, but it did not. Static timing analysis was used to ensure reliable operation at the chosen clock frequencies. Figure 5.2 shows the distribution of timing slack for circuit paths in the slow clock domain after the design was placed and routed. Fewer than 5% of circuit paths have less than 2 nanoseconds (ns) Worst Negative Slack (WNS). This suggests that manual placement or manual Register Transfer Level (RTL) optimization would allow a faster clock frequency in the slow domain.

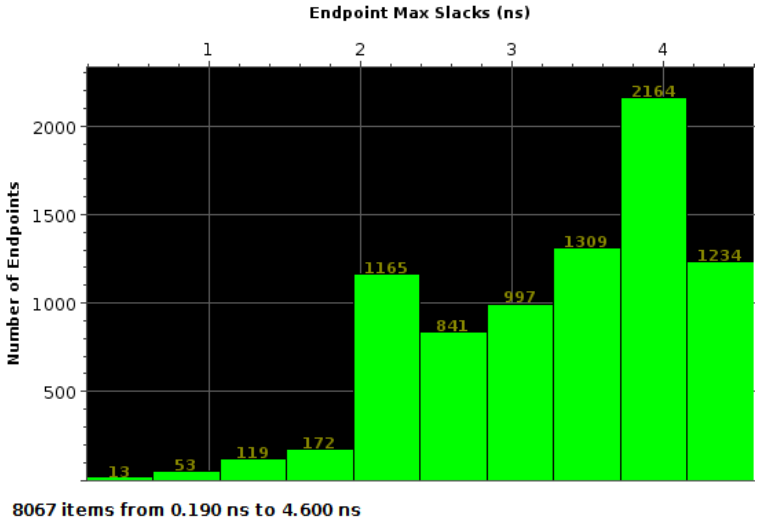

Figure 5.2: Distribution of circuit paths by Worst Negative Slack (slow domain).

The critical paths in the fast domain are due to long combinational logic paths in the instruction analyzer and the process detector. In the second case, additional pipeline registers would reduce path length. A more extensive redesign of the instruction analyzer may be needed to reduce logic depth. Figure 5.3 shows the distribution of timing slack for circuit paths in the fast clock domain.

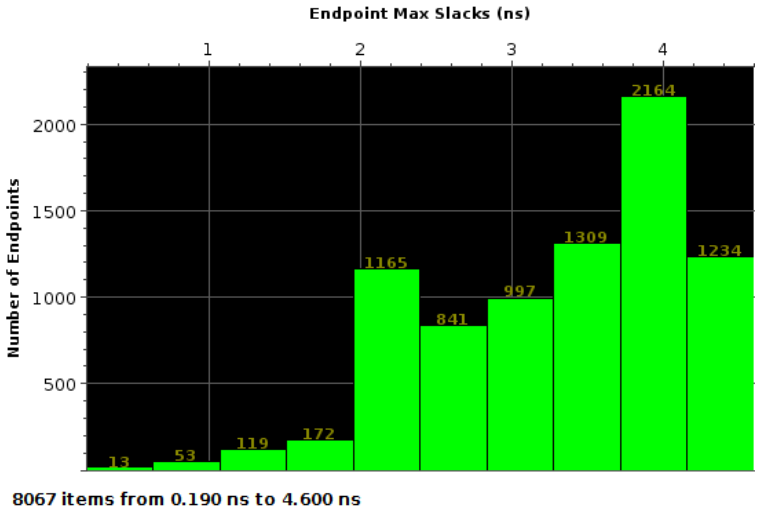

Figure 5.3: Distribution of circuit paths by Worst Negative Slack (fast domain).

Figure 5.4 shows the placement of FPGA resources on the FPGA. Resource utilization is shown in Table 5.2. Only a small fraction of FPGA resources are used. Most resources are available for use by other IP cores integrated into the same system. Arithmatic in the hash function unit is mapped to fixed DSP48E2 [20] arithmatic pipelines offerred by the programmable logic fabric. In many signal processing applications these units are valued highly so a system integrator would want to replace the hash function of HSMON with one that uses fewer arithmatic units or use time-domain multiplexing to share a single hash function module between both hash queue modules.

Power analysis was performed on the IP core using Xilinx recommended best practices [21]. Logic simulation of the post-synthesis netlist was used to measure circuit circuit switching activity to increase the accuracy of dynamic power estimation. Figure 5.5 shows the power summary for the IP core.

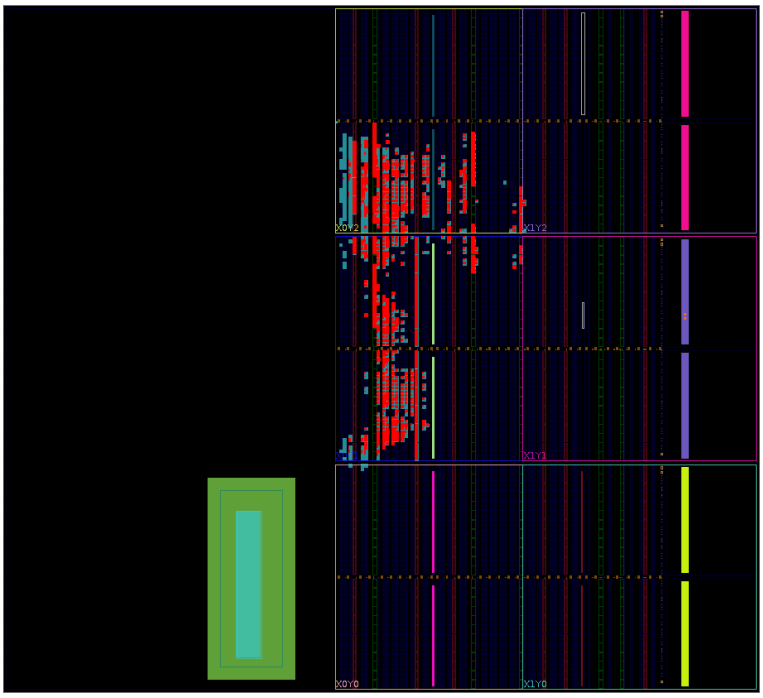

Figure 5.4: FPGA circuit placement (HSMON logic units in red).

5.2 Performance

The Dhrystone benchmark was run on our prototype system to detect processor core slowdown due to instruction tracing [22]. The benchmark program ran 50 million iterations of its testing algorithm. We repeated it with instruction tracing enabled and a second time with instruction tracing disabled. When tracing was disabled, we measured 3866976 Dhrystone per second. With tracing enabled, 3866976 Dhrystone per second were observed. The benchmark was also invoked through SMTool. In this case, 3875969 Dhrystone per second were observed. It is not clear why there is a 0.2 percent improvement in performance in this benchmark but it is consistent over multiple repetitions.

We also evaluated performance overhead due to tracing using the SPEC CPU 2006 benchmarks. Each benchmark was run five times with tracing enabled, and tracing disabled. We compare the arithmatic mean run time of each set of five to determine whether tracing has a measurable effect on the programs. Figure 5.6 shows the percentage change in mean run time for each benchmark. Small variations are due to the networked filesystem used by the prototype system. HSMON has no effect on CPU performance.

Table 5.2: FPGA resource utilization.

Resource	Utilization	Available	Utilization %
LUT	3041	47232	6.44
LUTRAM	176	28800	0.61
FF	3326	94464	3.52
BRAM	17	150	11.33
DSP	36	240	15.00
IO	2	252	0.79
BUFG	2	196	1.02

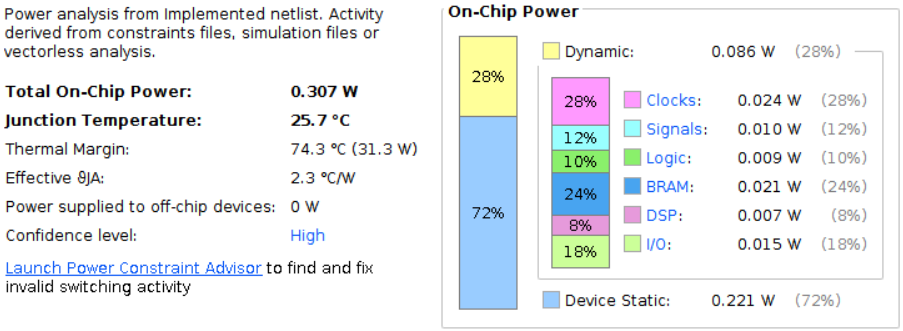

Figure 5.5: IP core power summary.

5.3 Attack Detection

We wrote three programs that allow attacks to overwrite indirect control flow targets by passing malformed command line arguments. Each program bug represents a typical target of a code-reuse attack. Two programs are vulnerable to stack buffer overflow, and one is vulnerable to heap buffer overflow. The stack buffer overflow programs are attackable by return address overwrite and function pointer overwrite. The heap program is attackable by C++ virtual table overwrite. The source code for each program is listed in Appendix E.

The first stack buffer overflow program (Listing E.1) allows an attack to overwrite the return address of the "main" subroutine. The 8-byte array in "act" is immediately below the frame pointer and the return address for the "main" subroutine on the stack. A successful attack on this program consists of passing a 24-byte string that "strcpy" writes over the frame pointer and return address.

The second stack buffer overflow program (Listing E.2) allows an attack to

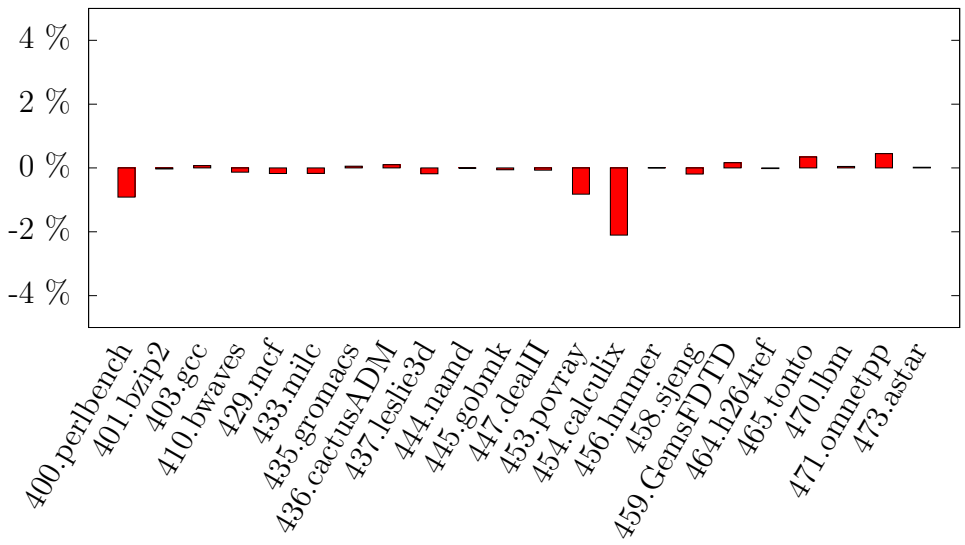

Figure 5.6: SPEC CPU 2006 change in run time.

overwrite a function pointer called by "main". The function pointer is stored immediately after the 8-byte array on the stack. A successful attack consists of passing a 16-byte string that overwrites the function pointer address.

The heap buffer overflow program (Listing E.3) allows an attack to overwrite the C++ virtual table pointer. The normal memory allocator stores the C++ object "obj" at a fixed offset from the array "buf" on the heap. Command line arguments allow the attacker to write arbitrary data to memory at offsets from the start of the array, which is allowed by our attack model.

For each program, we use the log feature of HSMON to learn the set of allowed signatures by providing safe command line inputs that cause the program to follow normal control flow paths. We then run each program with normal inputs to verify that HSMON does not detect a violation, and again with attack inputs to verify that HSMON counts violations that occur. Each program was allowed to run to completion with the attack input, so multiple violations can occur.

Table 5.3 shows the results of testing the three attackable programs. In each case, HSMON detects multiple violations because the programs are allowed to continue to completion.

Table 5.3: Attack detection results.

Program	Attack	Nr. System Call	Nr. Violation
Stack-Return (E.1)		3	0
Stack-Return (E.1)	✓	16	14
Stack-F. Pointer (E.2)		3	0
Stack-F. Pointer (E.2)	✓	9	8
Heap-V. Table (E.3)		4	0
Heap-V. Table (E.3)	✓	12	8

CHAPTER 6

CONCLUSION

In this work, we presented a hardware design for detecting when a program makes unexpected indirect control flow transfers. We developed a new method we call indirect control path signatures to describe allowable control flow transfers in a program. A method was suggested for deriving signatures from programs through static analysis. A hardware implementation was designed, prototyped, and evaluated in a test system.

6.1 Future Work

Future work needs to develop a theoretical foundation to provide accurate security guarantees using indirect control path signatures. We believe that the combination techniques presented in this work present difficulties to an attacker but the limits of this protection should be explored.

Improved software tools are needed. Sophisticated static analysis for signature generation from binary programs improves the usefulness of HSMON. The hardware IP core can monitor any software running on our prototype platform but it is of limited usefulness without a way to extract signatures from large programs. Software support is needed to support dynamic libraries, just-in-time compiled code, and run-time dynamic linking (dlopen). Operating system drivers need to be developed to receive interrupts, kill tasks, and support context switching between multiple monitored programs.

In Section 5.1 we saw that the clock frequencies of the hardware design can be improved. The slow clock domain should operate at the maximum frequency of the TPIU (250 MHz) to maximize the available bandwidth between the ARM Trace Bus and the Programmable Logic. Future work supporting multiple cores will need higher performance across this interface. Increased frequency in the fast clock domain will allow a smaller clock-crossing FIFO

for bringing trace data into the fast domain.

FPGA resource utilization, also called area, can be optimized by moving the Length Decoder from the slow clock domain to the fast clock domain. As previously discussed, the critical timing path in the slow clock domain is in this module so this change would also increase the maximum frequency. Hash Function units could support multiplexed input and output to reduce the number of DSP units required. Alternate hashing functions that are more optimal for hardware area is another aspect to explore.

In this work there was no design effort to reduce power consumption. Power efficiency is the most important design consideration for 64-bit ARM application processors that are similar those in our prototype platform. FPGA programmable logic has relatively high static power consumption compared to ASIC CMOS circuits but improved logic design can narrow the gap. Power hungry block RAM for the FIFOs and Bloom filter can be powered off when they are not used. Due to high static power consumption, we expect that any area reduction of the IP core will also reduce power consumption.

6.2 Final Remarks

Many improvements can be made to the design presented here. We believe that the use of hardware support for instruction tracing offers new capabilities to the security research community. We hope to see more support added to next-generation SoC and CPU designs so that more researchers will explore the use of tracing to enable secure systems.

REFERENCES

[1] M. Abadi, M. Budiu, U. Erlingsson, and J. Ligatti, "Control-flow integrity," in *Proceedings of the 12th ACM Conference on Computer and Communications Security*, ser. CCS '05. New York, NY, USA: ACM, 2005. [Online]. Available: http://doi.acm.org/10.1145/1102120.1102165 pp. 340–353.

[2] R. Roemer, E. Buchanan, H. Shacham, and S. Savage, "Return-oriented programming: Systems, languages, and applications," *ACM Trans. Inf. Syst. Secur.*, vol. 15, no. 1, pp. 2:1–2:34, Mar. 2012. [Online]. Available: http://doi.acm.org/10.1145/2133375.2133377

[3] S. Checkoway, L. Davi, A. Dmitrienko, A.-R. Sadeghi, H. Shacham, and M. Winandy, "Return-oriented programming without returns," in *Proceedings of the 17th ACM Conference on Computer and Communications Security*, ser. CCS '10. New York, NY, USA: ACM, 2010. [Online]. Available: http://doi.acm.org.proxy2.library.illinois.edu/10.1145/1866307.1866370 pp. 559–572.

[4] R. de Clercq and I. Verbauwhede, "A survey of hardware-based control flow integrity (CFI)," *CoRR*, vol. abs/1706.07257, 2017. [Online]. Available: http://arxiv.org/abs/1706.07257

[5] K. Wilken and J. P. Shen, "Continuous signature monitoring: Efficient concurrent-detection of processor control errors," in *International Test Conference 1988 Proceeding: New Frontiers in Testing*, Sep 1988, pp. 914–925.

[6] M. A. Schuette and J. P. Shen, "Processor control flow monitoring using signatured instruction streams," *IEEE Transactions on Computers*, vol. C-36, no. 3, pp. 264–276, March 1987.

[7] M. Zhang and R. Sekar, "Control flow integrity for COTS binaries," in *22nd USENIX Security Symposium (USENIX Security 13)*. Washington, D.C.: USENIX, 2013. [Online]. Available: https://www.usenix.org/conference/usenixsecurity13/technical-sessions/presentation/Zhang pp. 337–352.

[8] S. Designer, "Getting around non-executable stack (and fix)," 1997. [Online]. Available: http://seclists.org/bugtraq/1997/Aug/63

[9] T. Bletsch, X. Jiang, V. W. Freeh, and Z. Liang, "Jump-oriented programming: A new class of code-reuse attack," in *Proceedings of the 6th ACM Symposium on Information, Computer and Communications Security*, ser. ASIACCS '11. New York, NY, USA: ACM, 2011. [Online]. Available: http://doi.acm.org/10.1145/1966913.1966919 pp. 30–40.

[10] Y. Lee, I. Heo, D. Hwang, K. Kim, and Y. Paek, "Towards a practical solution to detect code reuse attacks on arm mobile devices," in *Proceedings of the Fourth Workshop on Hardware and Architectural Support for Security and Privacy*, ser. HASP '15. New York, NY, USA: ACM, 2015. [Online]. Available: http://doi.acm.org/10.1145/2768566.2768569 pp. 3:1–3:8.

[11] Y. Lee, J. Lee, I. Heo, D. Hwang, and Y. Paek, "Integration of rop/jop monitoring ips in an arm-based soc," in *Proceedings of the 2016 Conference on Design, Automation & Test in Europe*, ser. DATE '16. San Jose, CA, USA: EDA Consortium, 2016. [Online]. Available: http://dl.acm.org/citation.cfm?id=2971808.2971884 pp. 331–336.

[12] *Zynq UltraScale+ MPSoC Data Sheet: Overview*, Xilinx, 7 2017, v1.5.

[13] *AMBA AXI Protocol Version 2.0*, ARM Limited, 3 2010, issue C.

[14] *ARM CoreSight Architecture Specification v2.0*, ARM Limited, 9 2013, issue D.

[15] *Zynq UltraScale+ Device Technical Reference Manual*, Xilinx, 12 2017, v1.7.

[16] *CoreSight Components Technical Reference Manual*, ARM Limited, 7 2009, issue H.

[17] *ARM Embedded Trace Macrocell Architecture Specification*, ARM Limited, 2 2016, issue D.

[18] *ARM Architecture Reference Manual*, ARM Limited, 12 2017, issue C.a.

[19] B. H. Bloom, "Space/time trade-offs in hash coding with allowable errors," *Commununications of the ACM*, vol. 13, no. 7, pp. 422–426, July 1970. [Online]. Available: http://doi.acm.org/10.1145/362686.362692

[20] *UltraScale Architecture DSP Slice User Guide*, Xilinx, 6 2018, v1.7.

[21] *Vivado Design Suite User Guide - Power Analysis and Optimization*, Xilinx, 7 2017, v2017.2.

[22] R. P. Weicker, "Dhrystone: A synthetic systems programming benchmark," *Commun. ACM*, vol. 27, no. 10, pp. 1013–1030, Oct. 1984. [Online]. Available: http://doi.acm.org/10.1145/358274.358283

APPENDIX A

HSMON REGISTERS

Programmable registers and register fields are described in Tables A.1–A.3. For more information about programming HSMON refer to Section 4.2.

Table A.1: Memory-mapped registers.

Offset	R/W	Name	Description
0	RW	Control	Bit flags to control operation of HSMON.
4	R	Status	Status bit flags to indicate event counter overflow.
68	R	Syscall Count	Number of system calls observed since reset.
72	R	Exception Count	Number of exceptions observed since reset.
76	R	Violation Count	Number of system calls that failed to match in the Bloom filter.
80	R	Drop Count	Number of 32-bit words dropped at the SF FIFO due to high trace data volume.
84	R	All Count	Number of 32-bit words received from the TPIU.
128	RW	Context ID	Context ID of the process to monitor. Normally the same as the process ID in Linux.

Table A.1: Memory-mapped registers (cont.).

Offset	R/W	Name	Description
136	RW	Start Address Low	Least significant 32-bit component of the 64-bit program start address.
140	RW	Start Address High	Most significant 32-bit component of the start address.
144	RW	Seed	Seed value for the MurmurHash3 hash algorithm.
148	RW	Seed 2	Seed value for the MurmurHash3 hash algoritm in the second function unit.
512	RW	Bloom Filter Address	The next read or write to the Bloom filter data register will access this location in the Bloom filter data memory.
516	RW	Bloom Filter Data	A read or write to this register accesses the Bloom filter memory at the address pointed to by the Bloom filter address register. The address register is incremented after the access.
520	RW	Bloom Filter Tested Data	A read or write to this register accesses the Bloom filter tested memory at the address pointed to by the Bloom filter address register. The address register is incremented after the access.

Table A.2: Control register bit fields.

Bit	Name	Description
0	ENABLE	HSMON enable flag. A transition from 0 to 1 on this bit enables the process detector.
1	RESET	Resets all internal HSMON state except for the trace decoder. Clears all memory mapped registers with an offset greater or equal to 128.
2	LOCK	Setting this bit disables read and write of memory mapped registers with an offset greater or equal to 128. Cannot be toggled to 0. This bit is cleared by toggling RESET.
3	READONCE	Settings this bit forces event counters to atomatically cleared to 0 during a read from the memory-mapped bus interface.
4	TWOHASH	Setting this bit enables the second hash function unit that uses seed 2.
5	TWOBIT	Setting this bit remaps the hash output to Bloom filter mapping so that at most two bits will be used per signature.
30	FERESET	Resets internal state of the trace decoder.
31	INTERRUPT	Interrupt service flag.

Table A.3: Status register bit fields.

Bit	Name	Description
5	SYSCALLOF	Syscall counter overflow flag.
6	EXCEPTOF	Exception counter overflow flag.
7	VIOLATEOF	Violation counter overflow flag.
8	DROPOF	Drop counter overflow flag.
9	ALLOF	All counter overflow flag.

APPENDIX B

HSMON PROGRAMMING SEQUENCE

The recommended programming sequence for HSMON is:

1. Clear the ENABLE bit in the control register.
2. Set the RESET bit in the control register.
3. If it is necessary to change the filter ID register, use the following sequence:
   (a) Set the FERESET bit in the control register.
   (b) Write the filter ID register.
   (c) Clear the FERESET bit.
4. Set the seed, context ID, and start address registers.
5. Set the TWOHASH and TWOBIT control registers according to the Bloom filter size and hash policy.
6. Initialize the Bloom filter address register to 0.
7. Write a the sequence of 2048 Bloom filter 32-bit vectors to the Bloom filter data register. If TWOHASH is 0, only 8 vectors need to be written.
8. Set the Bloom filter address register to 0.
9. If the Bloom filter log will be used, write zero to the Bloom filter tested data register 2048 times. If TWOHASH is 0, it only need written 8 times.
10. Set the Bloom filter address register.
11. Set the ENABLE bit in the control register.

APPENDIX C

SMTOOL HWDAT FILE FORMAT

HWDAT is the file format used by SMTool to save and load memory-mapped registers from the filesystem. There are two formats, selected by the 4th bit in the file. In Table C.1 the column TH0 Off lists the byte offset for a field when the 4th bit is 0, and the column TH1 Off lists the byte offset for the field when the 4th bit is 1.

Table C.1: HWDAT file format.

Field	TH0 Off	TH1 Off	Description
Control Register	0	0	Control register value. Only bits 4 and 5 are loaded.
Seed	4	4	Seed register value.
Seed2		8	Seed 2 register value.
Start Address Low	8	12	Start address low register value.
Start Address High	12	16	Start address high register value.
BFilter 0	16	20	Bloom filter memory address 0 value.
BFilter 1	20	24	Bloom filter memory address 1 value.
BFilter 2	24	28	Bloom filter memory address 2 value.
⋮			
BFilter 2047	8204	8208	Bloom filter memory address 2047 value.

APPENDIX D

SMTOOL UNIX MANUAL PAGE

NAME

smtool − control the signature monitor IP core

SYNOPSIS

smtool [**−rwsSTmg**] [**−a** *ADDR*] [**−x** *cmd...*]

DESCRIPTION

cstool accesses the memory-mapped interface of the signature monitor IP core through /dev/mem.

OPTIONS

−r Read from the register at ADDR.

−w Write DATA to the register at ADDR.

−a *ADDR*
Sets the 32-bit word address to read or write in hexadecimal.

−d *DATA*
Sets the data to read or write in hexadecimal.

−s Print a summary of register settings.

−S Print a summary of register settings with memory dump of the Bloom Filter memory.

−T Also print a memory dump of the Bloom Filter log. Must be used with **−s** or **−S**.

−x *cmd...*
Execute *cmd* as a child process after the signature monitor hardware is configured. Requires a*cmd*.hwdat file.

−g Save a copy of the register and memory values to the file *cmd*.hwdat.out. Must be used with **−x** *cmd*.

−m Merge the Bloom Filter log with the Bloom Filter memory before saving. Must be used with **−x** *cmd* and **−g.**

BUGS

The options [**−gm**] should not depend on **−x**.

AUTHOR

Eric Clark <ejclark2@illinois.edu>

APPENDIX E

EXPLOIT EXAMPLE PROGRAM CODE

Listing E.1: Stack buffer overflow program for return address overwrite.

```
#include <string.h>
#include <stdio.h>
#include <stdlib.h>

void dead()
{
        system("echo hi");
}

void act(char *str)
{
        char name[8] = {0};

        strcpy(name, str);
        printf("hello %s\n", name);
}

int main (int argc, char **argv, char **env)
{
        if (argc < 2)
                return 1;

        act(argv[1]);

        return 0;
}
```

Listing E.2: Stack buffer overflow program for function pointer overwrite.

```
#include <string.h>
#include <stdio.h>
#include <stdlib.h>

typedef int (*fptr)(char *);

int deadcode(char *name)
{
        system(name);
}

int pp_a(char *name)
{
        printf("hello %s\n", name);
}

int pp_b(char *name)
{
        printf("goodbye %s\n", name);
}

static fptr fnlist[2] = {pp_a, pp_b};

int main (int argc, char **argv, char **env)
{
        char name[8] = {0};
        fptr fn;

        if (argc < 2)
                return 1;

        fn = fnlist[argc-2];

        strcpy(name, argv[1]);
        fn(name);

        return 0;
}
```

Listing E.3: Heap buffer overflow program for virtual table overwrite.

```
#include <cstdio>
#include <cstring>
#include <cstdlib>

class Base {
        public:
        char ch;

        virtual void f() = 0;
        void p() { printf("%c::f()\n", ch); }
};
class A : public Base {
        public:
        virtual void f() { ch = 'A'; }
};
class B : public Base {
        public:
        virtual void f() { ch = 'B'; }
};
class C : public Base {
        public:
        virtual void f();
};
void C::f() { ch = 'C'; }
int (*fptr)(const char*) = system;

int main (int argc, char **argv, char **env)
{
        int i;
        Base *obj;

        if (argc < 4)
                return 1;
        printf("\n");

        unsigned long *buf = new unsigned long[16];
        if (!atoi(argv[1])) {
                obj = new A();
        } else {
                obj = new B();
        }
        buf[atoi(argv[2])] = strtoul(argv[3], NULL, 16);

        obj->f();
        obj->p();

        delete obj;
        delete buf;

        return 0;
}
```